Learning For Self-Driving Cars and Intelligent System
Truck Platooning

Changyao Zhou*, Jiaxin Pan†,
Robotics, Cognition, Intelligence
Technical University of Munich, Munich, Germany
Email: *changyao.zhou, †jiaxin.pan@tum.de

***Abstract*—Truck platooning is a hot research topic these days since it brings benefits in different fields. In this project, we combined truck platooning with model predictive control (MPC). With MPC outputs as ground truth, we train different neural networks to predict control values. To improve the possibility to be implemented in reality, we tried models with different inputs and datasets step by step. Most of our resulting models was robust for random impulses or multiple following vehicles.**

***Index Terms*—MPC, car platooning, CARLA[1], deep learning**

I. Introduction

Truck platooning is an important innovation in the automotive industry that aims at improving the safety, mileage, efficiency, and time needed to travel [1]. The topic aims at enabling multiple trucks to drive along the same trajectory while a small gap is kept in between, with only one driver in the first truck. The control values of the following trucks are predicted by the model and little action is required from the driver in the first truck. The concept is shown in Figure 1 [2].

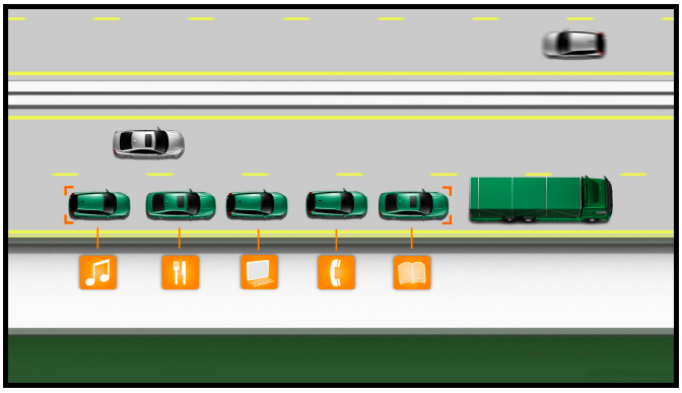

Figure 1. Platooning Illustration

Truck platooning has been a hot research topic for years since several advantages have been proved with platooning. With only a short gap kept between the vehicles, less air friction is applied to the vehicles, resulting in less fuel consumption. What's more, since the following vehicles are controlled automatically instead of by human drivers, less reaction time is needed to adapt to the movements of the leading vehicle, which improves safety. Apart from less emission and improved safety, transportation can be optimized by using roads more effectively due to the shorter gap between vehicles.

Model predictive control (MPC) is an advanced method of process control while satisfying a set of constraints and has been proved feasible as for truck platooning [3], [4]. Thus, MPC is used in this project for generating ground-truth control values of the following vehicles (ego vehicles).

To combine modern deep learning with truck platooning, several neural networks are built with different architectures and trained to predict the control values for the ego vehicles.

The training outputs are tested in the CARLA simulator [5], which helps to simulate the real world. The result videos can be seen in the lrz Folder[2].

II. Problem Statement

In this project, we focused on training a neural network that predicts control values for the ego vehicle, including throttle and steering angle, using outputs from the MPC controller as ground truth. The ego vehicle is designed to follow a target vehicle and keep a proper distance in between.

The goal of this project is to design a framework that can be realized in the real world and the CARLA simulator v0.9.11 is used to test our model.

Our project contains the following steps:

1) Control the ego-vehicle with Model Predictive Control (MPC) in the CARLA simulator
2) With the control values from MPC as ground truth, train a neural network to control the ego-vehicle by using the MPC output as ground truths
3) Further improvements by changing the inputs of the neural network, making our model closer to the real world

In this report, the implementation of MPC is introduced in section III. Four different neural networks that are closer to the reality step by step are introduced in detail in sections IV, VI and VII. Finally, the evaluation metrics and results are discussed in VIII.

[1]https://carla.org/

[2]https://syncandshare.lrz.de/getlink/fiFyLGH5PvnfGPKorSXtX8x7/Final%20Result%20Truck%20Platooning

III. MODEL PREDICTIVE CONTROL

In this section, the concept of model predictive control (MPC) is first briefly introduced. The implementation of MPC in this project is then discussed in detail.

A. Introduction to MPC

MPC is an advanced method of process control that is used to control a process while satisfying a set of constraints. It takes a prediction horizon instead of a single time step into account and aims to get an optimal control result by minimizing the cost function within the prediction horizon.

B. Implementation of MPC

The MPC is used to generate optimal control values for the ego vehicle which can satisfy constraints such as range of the control values and meanwhile minimize the cost function regarding car distance. The implementation is shown in Figure 2

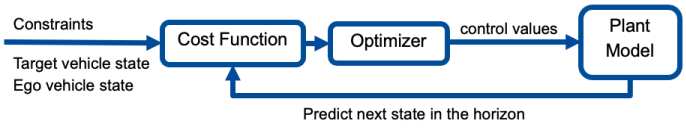

Figure 2. Implementation of MPC in this project.

C. The Offline-mode Method

We first tried to implement our MPC model in an 'offline mode', which means we first run the CARLA simulator only with one target vehicle controlled by autopilot and wrote down its positions, velocities, yaw angles, and control values. Then we used our MPC model to generate the control values for ego-vehicle offline, taking the trajectory of the target within the following 15-time steps into account. After that, we run the CARLA simulator again with both the target vehicle and ego vehicle, with the target vehicle controlled by the control values collected in the last time and the ego vehicle controlled by the output from the MPC.

This method managed to control the ego vehicle most of the time, but is not practical in reality, since the information of the target vehicle is not reachable in advance. Thus, the offline-mode method was taken place with the online-mode method, which is described in the next subsection.

D. The Online-mode Method

Since the offline-mode method is not achievable in practice, it is replaced by the online-mode method. With this method, instead of running the MPC to generate control values with recorded information of the target vehicle for the whole trajectory, the positions, yaw angles, and velocities of both target and ego vehicles are detected in each time step when running the CARLA simulator online and the control values are computed based on these real-time inputs with MPC. With this 'online-mode' method, only information at the current time step and in the past are available, which is the same as in the real world.

Following are the cost function and the plant model in our MPC model.

E. The Cost function

The MPC optimizes the control values by minimizing the cost function in the prediction horizon, which is set as 10 in our model.

Since both target and ego vehicles are assumed to be moving on the horizon, their state needs to be updated at each time step. The state of the ego vehicle is updated with the plant model, which is introduced in the next subsection. The velocity and yaw angle of the target vehicle is assumed unchanged within the horizon since the throttle and steering angle are unknown.

Thus, the state of the target vehicle is updated in each step within the horizon with the following equations:

$$\begin{aligned} x_{ref} &= x_{ref} + v_{ref} * cos(yaw_{ref}) * dt \\ y_{ref} &= y_{ref} + v_{ref} * sin(yaw_{ref}) * dt \end{aligned} \tag{1}$$

$x_{ref}, y_{ref}, v_{ref}, yaw_{ref}$ represent the state of the ego vehicle.

With the updated states of the target and ego vehicle, the cost is computed in each step based on their states. The cost is mainly divided into three parts: the position cost, the angle cost, and the velocity cost.

1) The Position Cost: To follow the target car, there must be a proper distance between the two vehicles to avoid the collision especially when there is a sudden stop, which is called the safe distance. The safe distance is designed based on the velocity of the target vehicle:

$$S = L * (1 + v_{target}/5) \tag{2}$$

where S represents the safe distance and L is the length of the vehicle. The parameter 5 is chosen since the speed of the target vehicle reaches the balance point with 5 m/s, based on the data we collected, 80% of the time the target vehicle drives with this speed. With the equation 2, the safe distance should be twice the length of the vehicle when the target vehicle has a constant velocity of 5 m/s.

The desired distances on x-/y-axis are computed based on the safe distance S respectively:

$$\begin{aligned} Dis_x &= S * (cos(yaw_{ego}) * 0.5 + cos(yaw_{target}) * 0.5) \\ Dis_y &= S * (sin(yaw_{ego}) * 0.5 + sin(yaw_{target}) * 0.5) \end{aligned} \tag{3}$$

The distance is a combination of projections with two yaw angles. The reason is shown in the Figure 3, with only yaw_{ego} the ego vehicle may take a shortcut, while with only yaw_{target} the turning radius may be larger as the target vehicle.

The position cost is then calculated separately for x-and y-axis with absolute distance based on the desired distance:

$$\begin{aligned} cost_x &= (abs((x_{ref} - Dis_x) - x_{ego})) \\ cost_y &= (abs((y_{ref} - Dis_y) - y_{ego})) \end{aligned} \tag{4}$$

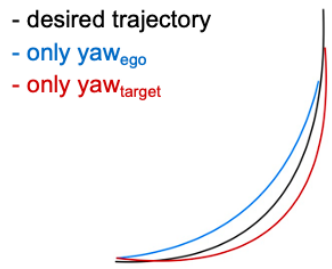

Figure 3. Trajectories with only one yaw angle

2) The Angle Cost: By vehicle following, the difference between the target vehicle and ego vehicle should be small.

The yaw angle in the CARLA simulator can not be matched perfectly to our simple model. The output yaw angle from the CARLA simulator has a range of [-π, π], while the more common range is [0, 2π]. This may lead to a problem that a very sharp turn around 360° is detected when the yaw of the target vehicle changes from -179.5° to 179.5° when computing the angle difference with direct subtraction, which is only a slight turn of 1°.

To solve this problem, we converted the range of the output yaw angle from CARLA simulator to the range [0, 2π] with the equation (2):

$$yaw_{ref} = (yaw_{ref} + 2 * pi)\%(pi * 2) \tag{5}$$

This lead to however another problem, for the original yaw angle -0.7°, it is converted to 359.3°, and for 0.1° it is converted to 0.1°. If the yaw of the target vehicle changed from -0.7° to 0.1°, a very sharp turn with around 360° is detected, while the actual turn is only 0.8°. If the current yaw of ego vehicle is -0.7°, an extremely large cost will be added, which is unreasonable. This is solved by modifying the cost function of the yaw angle:

$$\begin{aligned} yaw_{ego} &= abs(yaw_{ego} - pi) \\ yaw_{ref} &= abs(yaw_{ref} - pi) \\ cost &= (yaw_{ego} - yaw_{ref})^2 \end{aligned} \tag{6}$$

With equation 6, the difference between the yaw angles of both vehicles is no longer computed with direct subtraction. Instead, first, their distance to 180° is computed separately and then the difference between these two distances is computed. For 359.3°, its distance to 180° is 179.3° and for 0.1° the distance is 180.1°. Therefore, the cost is reduced to 0.8°, which is much more reasonable.

3) The Velocity Cost: The last part of the cost function is based on the velocities of both vehicles:

$$cost = abs(v_{ref} - v_{ego}) * 2 \tag{7}$$

F. The Plant Model

The plant model is used to update the state of the ego vehicle within the prediction horizon and the parameters are chosen to match the motion of the vehicle in the CARLA simulator. The model introduced in an online course[3] is taken as the inference.

$$\begin{aligned} x_{t+1} &= x_t + v_t * dt * cos(yaw_t) \\ y_{t+1} &= y_t + v_t * dt * sin(yaw_t) \\ yaw_{t+1} &= yaw_t + v_t * dt * beta/length \\ v_{t+1} &= v_t + a_t * dt - v_t/19.8 \end{aligned} \tag{8}$$

x_t, y_t, v_t, yaw_t represent the current state of the ego vehicle, while $x_{t+1}, y_{t+1}, v_{t+1}, yaw_{t+1}$ are the predicted next state. a_t and $beta$ are computed based on throttle and steering angle respectively, which are optimized by the MPC.

It is proved that this model can not predict the exact motion of the ego vehicle in the CARLA simulator, mainly because the motion model for each gear level is different in the CARLA simulator, while only one plant model is used in our MPC and the gear is not taken into account.

Though there is a difference between the plant model and the motion model in the CARLA simulator, the plant model can help to predict the future states of the ego vehicle roughly.

G. Constraints of Control Values

In addition to the cost function and plant model, the reasonable range of the two control values is also considered while optimizing. The range of throttle is [0, 1], and the range of steering angle is [-1, 1]. Now we can compute the optimal solution with *scipy.optimize*, taking the cost function, plant model, and constraints into account.

H. Test Result with MPC

The MPC is tested by predicting the control values for the ego vehicle with our MPC in the CARLA simulator, using the online-mode method.

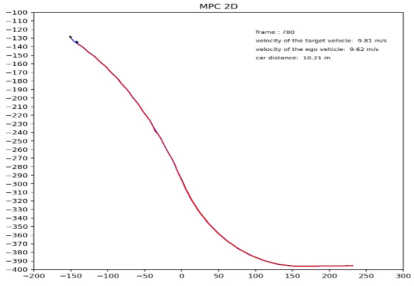

Figure 4. Testing result with MPC.

The result is shown in Figure 4, where blue dots represent the trajectory of the target vehicle and the red dots represent the trajectory of the ego vehicle. As can be seen from the figure, the two trajectories overlap almost at every step. This means that the control values generated from the MPC are quite desirable.

[3]https://github.com/WuStangDan/mpc-course-assignments

IV. MLP WITH RAW STATES

In this section, the very first model MLP with raw states is introduced in detail.

A. Data Collection

For data collection of this beginning model, the online-mode method, which is introduced in section III-D, is used. Only one target vehicle and one ego vehicle are created for data collection since all the following vehicles should be controlled with the same model. When running the CARLA simulator, a target vehicle is created with a random color, controlled with autopilot, and the ego vehicle is controlled by control values generated by MPC. In each frame, x-/y-coordinates, yaw angles, and velocities of both vehicles are written down, which are the so-called raw states and are used as input of this MLP model. The corresponding control values generated from MPC in each frame are written down as well as ground-truth outputs for training the neural network. The architecture is shown in Figure 5

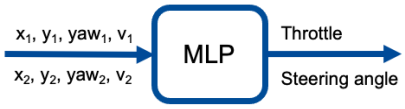

Figure 5. Architecture of MLP model with raw states

Since the network-output control values may not be very perfect and may lead the ego vehicle to unexpected positions, besides the 'on-trajectory' data, shown as blue dots in Figure 6, where the ego-vehicle follows the target vehicle as designed, some 'off-trajectory' data are also collected, which are control values for the positions neighboring the ideal trajectory, shown as red dots in Figure 6. As can be seen in the figure, almost all possible positions around the on-trajectory data are covered during data collection. This method helps the network to learn how to react when the ego-vehicle is not following the target vehicle perfectly.

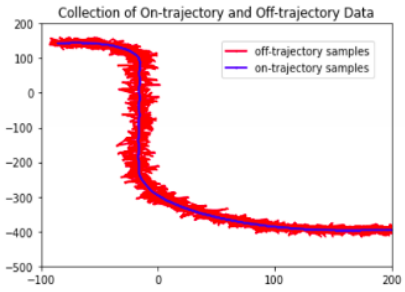

Figure 6. On-trajectory and off-trajectory data

To collect the off-trajectory data, two methods are tried, namely the random-impulses method and the random-position method, which are introduced in the following subsections in detail.

1) Random-impulses Method: The random-impulses method aims at applying random impulses to the ego vehicle and pushing it to off-trajectory regions. To apply random impulses on the ego vehicle, the Carla.actor.add_impulse[4] is used. Both the direction and the magnitude of the impulses are chosen randomly. The direction is set according to the global axis and can be either +x, -x +y, or -y. As shown in equations 9, the magnitude of the impulse is chosen from a range, which is computed based on the mass of the vehicle. Parameters 4 and 8 are chosen to push the ego vehicle for a reasonable distance based on experiments.

$$\begin{aligned} physics\_vehicle &= vehicle.get\_physics\_control() \\ car\_mass &= physics\_vehicle.mass \\ impulse &= random.uniform(4.0, 8.0) * car\_mass \\ vehicle.add\_impulse(carla.Vector3D(impulse, 0, 0)) \end{aligned} \tag{9}$$

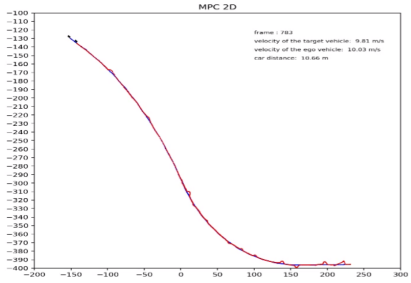

Figure 7. Off-trajectory data collection with random impulses

The test result can be seen in Figure 7. The MPC managed to generate reasonable control values that lead the ego vehicle back to the ideal trajectory. However, since it is not common in the reality that the vehicle is applied with impulses and it costs extra frames to push the vehicle in comparison to the random-position method, this method is replaced with the random-position method.

2) Random-position Method: With this method, instead of applying impulses to the ego vehicle, a set of points around the ideal trajectory of the ego vehicle are sampled during the collection, so-called random positions, while the velocities and yaw angles of the ego vehicle are also changed as disturbances. Then the disturbed ego-vehicle should go back to follow the target vehicle with help of the MPC controller. The test result is shown in Figure 8. In comparison to the random-impulse method, in which only a few off-trajectory positions are covered, much denser are the random positions around the ideal trajectory, shown as red dots in the figure.

Since there are disturbances not only on positions but also on velocities and yaw angles of the ego vehicle, almost all possible situations are taken into account, enabling the network to deal with most of the unexpected trajectories.

B. Architecture and Result

With the random-position method, nearly 30,000 off-trajectory data are collected around one trajectory in town 04 with 800

[4]https://github.com/carla-simulator/carla/blob/8854804f4d7748e14d937ec763a2912823a7e5f5/Docs/python_api.md#method

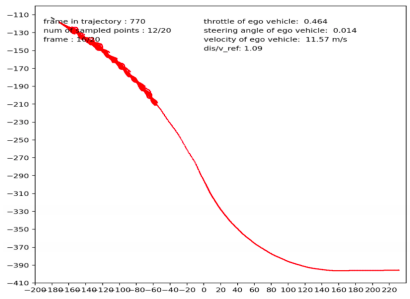

Figure 8. Off-trajectory data collection with random positions

Table I
ARCHITECTURE FOR THE MLP MODEL WITH RAW STATES

Layer Number	Layer Type	Layer Input	Layer Output
1	Linear Layer	8	256
2	ReLU	256	256
3	BatchNorm	256	256
4	Linear Layer	256	1024
5	ReLU	1024	1024
6	BatchNorm	1024	1024
7	Linear Layer	1024	2048
8	ReLU	2048	2048
9	BatchNorm	2048	2048
10	Linear Layer	2048	1024
11	ReLU	1024	1024
12	BatchNorm	1024	1024
13	Linear Layer	1024	256
14	ReLU	256	256
15	BatchNorm	256	256
16	Linear Layer	256	2

frames. Inputs of the model contain corresponding raw states of both vehicles, which include 8 elements, outputs are the control values, namely throttle and steering angle.

A 6-layer MLP model is designed for training, combining with ReLU activation and Batch normalization, whose architecture is shown in Table I.

After training for 6,000 epochs with the learning rate 1e-4 and 1e-6, the average validation loss reached 9e-2. The outputs of this model were tested in the CARLA simulator, where the ego vehicle tried to follow the target vehicle on the seen trajectory with control values predicted by the MLP model while being impulsed randomly with the function introduced in 9.

The test video can be seen in the supplementary link, in the folder */MLP with raw states/*. According to the test, the ego vehicle managed to follow the target vehicle while keeping a reasonable distance in between even far from the ideal trajectory due to random impulses.

V. MLP WITH RELATIVE STATES

A. Overview for MLP model with relative states

Since the testing result with the first MLP model was quite ideal, it is proved that a simple MLP model is capable to predict control values accurately with raw states as input information. In the real world, however, the accurate raw states may not be available since they can not be detected directly by sensors especially the global x-/y- coordinates and yaw angles.

Therefore, instead of raw states, relative states and the velocities are used as inputs of this second MLP model, which are more easily reachable. The architecture is shown in Figure 9

Figure 9. Architecture of MLP model with relative states

The relative states, containing $\Delta x, \Delta y, \Delta yaw$ are computed based on the dataset containing raw states collected in the first MLP model. Besides the relative states, the original velocities v_{target} and v_{ego} are appended to the relative states, forming an input with 5 elements of this second MLP model. The two velocities are not converted to Δv, since this might lead to the different ground truth for the same inputs. When vehicles drive with different constant speeds on different trajectories, which outputting 0 for Δv, the throttles of the ego vehicles differ due to different constant speeds.

Another different input structure, containing $\Delta x, \Delta y, yaw_{target}, yaw_{ego}, v_{target}, v_{ego}$ was also tried for training and produced relatively reasonable result. However, since this input structure requires raw information of yaw_{target} and yaw_{ego}, which are not always available in reality, this is abandoned.

Two methods are used to compute the relative states, namely the direct-subtraction method and the relative-coordinate method. In the following subsections, these two methods are discussed in detail.

B. Direct-Subtraction Method

The first method used to compute the relative states is the so-called direct-subtraction method, with which the relative states equal to target raw states minus ego raw states, the equations are shown as followings:

$$\begin{aligned} \Delta x &= x_{target} - x_{ego} \\ \Delta y &= y_{target} - y_{ego} \\ \Delta yaw &= yaw_{target} - yaw_{ego} \end{aligned} \tag{10}$$

This method is a relatively simple prepossessing method based on the raw states but may result in unreasonable relative states. Examples are shown in Figure 10. In this figure, two example trajectories towards different directions are shown, which are colored in blue and grey respectively, where the darker circles represent the target vehicles. With the blue vehicles driving towards +x direction, the resulting relative states, which contains $(\Delta x, \Delta y, \Delta yaw)$, should be $(7, 0, 0)$, while the resulting relative state of the grey vehicles, which driving towards +y direction, should be $(0, 7, 0)$. Assuming vehicles on both trajectories are running with a constant speed $5m/s$, the control values for both ego vehicles, which contains (throttle, steering angle) should be the same, which is roughly

$(0.35, 0)$. Since the relative states, which are used as inputs for the model, for two trajectories are different, while the control values for the ego vehicles, which are the ground-truth outputs for the model, are exactly the same, the model might be confused by training.

This problem is proved by training the neural network, many networks with different architectures are trained with this direct-subtraction method, all leading to much more unstable testing results in comparison to the first MLP model with raw states. Even tested with the trajectory in the training dataset, the ego vehicle could not follow the target vehicle for the whole trajectory.

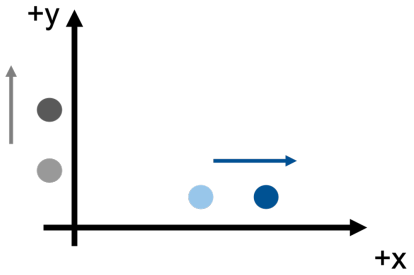

Figure 10. Examples for direct-subtraction method

C. Relative-Coordinate Method

To solve the problem of different inputs with the same outputs, the direct-subtraction method is replaced with the relative-coordinate method for computing relative states. With this method, the target-vehicle coordinate is re-computed with respect to the ego-vehicle coordinate. The transformation matrix from world to the two vehicles are first computed separately with:

$$\begin{aligned} Tr &= \begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} cos(\theta) & -sin(\theta) & x \\ sin(\theta) & cos(\theta) & y \\ 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{11}$$

where x, y, θ represents the x-/y-coordinate and yaw angle of the vehicle respectively. Then the transformation matrix of ego vehicle to target vehicle can be computed with:

$$\begin{aligned} Tr_2^1 &= \begin{bmatrix} R_1 & T_1 \\ 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} R_2 & T_2 \\ 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} R_1^T & -R_1^T T_1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} R_2 & T_2 \\ 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} R_1^T R_2 & R_1^T (T_2 - T_1) \\ 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \Delta R & \Delta T \\ 0 & 1 \end{bmatrix} \end{aligned} \tag{12}$$

where 1 represents the ego vehicle and 2 represents the target vehicle, ΔR and ΔT represent the rotation and translation difference between two vehicles and are used as part of the inputs for the MLP model. The resulting ΔT contains two elements, Δx, and Δy, with Δx in most cases positive, representing longitudinal distance and Δy representing lateral distance between two vehicles.

With this relative-coordinate method, the relative states are much more reasonable, solving the problem of the same outputs with different inputs, which leads to undesirable testing results for the model trained by the direct-subtraction method.

Table II
ARCHITECTURE FOR THE MLP MODEL WITH RELATIVE STATES

Layer Number	Layer Type	Layer Input	Layer Output
1	Linear Layer	5	128
2	ReLUg	128	128
3	BatchNorm	128	128
4	Linear Layer	128	256
5	ReLU	256	256
6	BatchNorm	256	256
7	Linear Layer	256	256
8	ReLU	256	256
9	BatchNorm	256	256
10	Linear Layer	256	256
11	ReLU	256	256
12	BatchNorm	256	256
13	Linear Layer	256	128
14	ReLU	128	128
15	BatchNorm	128	128
16	Linear Layer	128	2

D. Architecture and Result

With the random-position method, nearly 60,000 off-trajectory data are collected around multiple trajectories in town 04 that generated 4,500 on-trajectory data. First the raw states are collected, then the relative states are computed based on the raw states. The input of the model includes 5 elements, containing $\Delta x, \Delta y, \Delta yaw, v_{target}, v_{ego}$, the outputs are throttle and steering angle for the ego vehicle, using MPC outputs as ground truth.

A 6-layer MLP model is designed for training, combining ReLU activation and Batch normalization, whose architecture is shown in Table II. In comparison to the model trained with raw states, whose architecture is shown in TableI, the MLP model trained with relative states has a simpler architecture.

After training only for 350 epochs, the validation loss reached 1.7e-3 for throttle, 4e-3 for steering angle, and 2.85e-3 on average. This was 10 times smaller than the model trained with data computed by direct subtraction and smaller than the validation loss for the MLP model trained with raw states, whose architecture is even more complex and required much more training epochs. Therefore, it is proved that the relative states provide enough information for training in an understandable way, easier to learn for the network.

The test videos can be seen in the supplementary link, in the folder */MLP model with relative states/*. The model predicts reasonable control values, leading the ego vehicle to follow the target vehicle stably, even though the test trajectory is not included in the training dataset. The test video for the model trained with data computed by the direct-subtraction method

is also provided in the link as a comparison, the infraction count is much larger in this video, which is tested with the trajectory in the training dataset.

VI. FCNN End-to-end

A. Overview for FCNN End-to-end Model

It is proved in sections IV and V that a simple MLP model with raw states or relative states can predict desirable control values for the ego vehicle. However, even with relative states, multiple sensors are required for detecting information, which may not be always available in reality. To be more realistic, cameras are used instead of sensors for the following sections for capturing image information as inputs for the neural network.

First, the FCNN (Fully convolutional Neural Network) end-to-end model was tried, whose input contained one image from the RGB camera and velocities from both vehicles. The output is the control value, as shown in Figure 11. We have also tried with another FCNN end-to-end model which predicts not only the control values, but the relative states, $\Delta x, \Delta y, \Delta yaw$. The result showed that adding relative states to prediction does not help to improve accuracy of predicting control values. Therefore, we continued with the model that only predict control values.

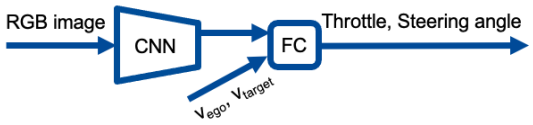

Figure 11. Architecture of CNN end-to-end model

Figure 12. Sample image from the RGB camera

Only one RGB image is used as input and a sample image is shown in Figure 12. The RGB camera with a field of view (FOV) of 100 ° is mounted on the front of the ego vehicle with almost the same height as the driver's view. With this height, even for the second or the third ego vehicle, only one vehicle ahead is captured in the image, instead of all the vehicles ahead. This enables predicting control values for several ego vehicles with the same model.

Table III
CNN End-to-end Architecture

Feature Extraction Module			
Layer Number	Layer Type	Layer Input	Layer Output
1 - 13	feature extractor from pretrained Alexnet	3x150x200	256x3x5
14	Flatten	258x3x5	3870
Prediction Module			
Layer Number	Layer Type	Layer Input	Layer Output
15	Linear Layer	3870	512
16	ReLU	512	512
17	Linear Layer	512	2

B. Image Prepossession

The RGB images were resized to 150x200 and prepossessed with image normalization, with which the pixels in the images were normalized to a range of [-1, 1] with:

$$torchvision.transforms.Normalize \\ ([0.485, 0456, 0.406], [0.229, 0.224, 0.225]) \tag{13}$$

This has a large impact on the training result and reduced the validation loss significantly.

C. Architecture and Training Result

With the random-position method, nearly 60,000 off-trajectory data are collected around multiple trajectories in town 04 that generated 4,500 on-trajectory data, which is exactly the same dataset as used for training V. Inputs of this model contains one RGB image with size 150x200 and the velocities, which are appended in the bottleneck.

The architecture is shown in Table III. The pretrained Alexnet is used for feature extraction, whose parameters are not frozen, and 2 extra linear layers are used trained for final prediction, as can be seen with layer numbers 15-17 in the Table. The velocities are appended between layers 13 and 14, before flattening.

After training for around 60 epochs, the validation loss reached 1.8e-2 on average, which was relatively small. And much less training epochs are required for training this model in comparison to MLP models

The test video can be seen in the supplementary link, in the folder */FCNN end-to-end model/*. The control values from the CNN end-to-end model were used to control the ego vehicle in the video, which only one RGB image and velocities for two vehicles as inputs. According to the test, the ego vehicle managed to follow the target vehicle while keeping a reasonable distance in between even for a sharp turn on an unseen trajectory.

VII. CNN-MLP model

From the section VI we can know, we have trained the end-to-end FCNN model and got a desirable result. However, we can't actually collect data online like what we did in CARLA simulator in practice. The reason is that we can't place the vehicle in a random position. It will definitely be dangerous and cause traffic accidents. The random position can be crossing the lane marking or in the opposite lane. Therefore, we can't collect data online in practice anymore.

Instead offline data collection is used now. In section IV we have introduced the on-trajectory data, which is exactly the trajectory of the target vehicle, and the off-trajectory data, which is the data collected from the region around the trajectory of the target vehicle. The basic idea of offline data collection is that we first collect the on-trajectory data in CARLA simulator and then generate the off-trajectory data outside the CARLA simulator. However, when we want to do so, there will be difficulty computing control values as ground truth, which requires accurate state updates for both vehicles. In online data collection, state update is automatically implemented by CARLA simulator and the real-time states of vehicles can be directly read from sensors. Different from online data collection, offline data collection without CARLA simulator needs an accurate plant model to update vehicle states. In fact, it is hard to obtain a precise plant model that can fit all trajectories. Therefore, we decide to divide the whole process of control value prediction into two stages and divide our model into two steps as well.

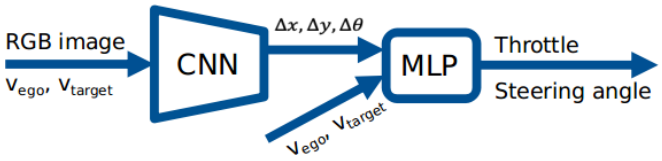

Figure 13. Architecture of Two-step Model

As shown in Figure 13, the previous end-to-end model is divided into two parts, a CNN (convolutional neural network) and a MLP (multi-layer perceptron). First CNN is used to predict the relative transformation between two vehicles. And then relative transformation, as well as two velocities, are fed into MLP to predict the control values. Since the same MLP model is trained in section V, so now our goal is to train a CNN which predicts the relative transformation.

A. CNN-MLP model with Online Dataset

To make sure the two-step model is reasonable for control value prediction, the model is first trained with the dataset directly collected from CARLA simulator. The model architecture is shown in Table IV , including the feature extractor of pretrained Alexnet [6] and several extra layers. After training for 100 epochs with Adam optimizer and learning rate 1e-4, the validation loss reaches the lowest point 1.8e-2. The losses for three prediction Δx, Δy, Δyaw are 4e-2, 1e-2, 1e-3 respectively. The inference of relative transformation is relatively accurate. Then we fed the prediction and two velocities of both vehicles into the MLP trained before. Since the dataset directly collected from CARLA simulator only contains two trajectories in Town04, the ego-vehicle controlled by the trained CNN-MLP model can generally follow the target vehicle around Town04. The test videos can be seen in the supplementary link, in the folder */CNN-MLP model with RGB camera/*. The test result seems good, which means the two-step model can be trained to predict control values.

Table IV
CNN ARCHITECTURE FOR RELATIVE TRANSFORMATION PREDICTION

Feature Extraction Module			
Layer Number	Layer Type	Layer Input	Layer Output
1 - 13	feature extractor from pretrained Alexnet	3x128x128	256x3x3
14	MaxPooling	256x3x3	256x1x1
15	Flatten	258x1x1	258
Control Values Prediction Module			
Layer Number	Layer Type	Layer Input	Layer Output
16	Linear Layer	258	64
17	ReLU	64	64
18	Linear Layer	64	3

B. Basic CNN-MLP Model

After the two-step model is proven to be able to predict control values, we first tried to train a basic model just for later comparison. It's trained only with a basic dataset, which only contains 16,000 on-trajectory samples from 8 trajectories in town 03, 04. After training the model for around 100 epochs, the validation loss can reach 2e-2 in total. The validation loss for Δx, Δy, Δyaw was 2.7e-2, 2.9e-2 and 1.6e-3 respectively. However, since the training dataset only contains on-trajectory samples, once the ego-vehicle deviates from the track of the target vehicle, it's hard for it to go back to the correct trajectory and it will get lost. Thus, the performance of the basic model is unstable in test trajectories in different towns. The test videos can be seen in the supplementary link, in the folder */CNN-MLP model with only on-trajectory data/*. Especially at sharp turns, the ego-vehicle controlled by a basic model tent to lose the target while following.

Then the next step is to implement offline data collection through automatically generating off-trajectory data. To bring the data collection process closer to reality, the following two approaches are used.

C. Depth-based CNN-MLP Model

To collect off-trajectory data, input images are generated through the image rendering process. Instead of square images with the camera FOV (field of view) 100° as before, RGB images and depth maps with a rectangular shape (800x300) and a larger FOV of 130° are collected for each on-trajectory sample. The images are collected through the sensor in CARLA simulator, RGB camera, and depth camera. Then at each on-trajectory sample, several off-trajectory points are sampled randomly around the on-trajectory point and the corresponding images as input and control values as ground truth are collected.

To collect off-trajectory data, longitudinal, lateral, and rotational offsets (x_0, y_0, θ_0) are added to on-trajectory data. The transformation matrix between ego-vehicle on the original trajectory and ego-vehicle with offset can be computed with

$$T_{ego}^{ego'} = \begin{bmatrix} cos(\theta_0) & 0 & sin(\theta_0) & y_0 \\ 0 & 1 & 0 & x_0 \\ -sin(\theta_0) & 0 & cos(\theta_0) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{14}$$

where x_0, y_0, θ_0 are longitudinal, lateral, and rotational offset added to ego-vehicle. Then the relative transformation

of target vehicle w.r.t. ego-vehicle with offset as ground truth is computed with

$$T_{target}^{ego'} = T_{ego}^{ego'} T_{target}^{ego} \tag{15}$$

where T_{target}^{ego} is the transformation of the ego-vehicle on the original trajectory w.r.t target vehicle, which can be computed with the states of both vehicles collected from CARLA simulator.

In addition, T_{target}^{ego} can be also used for off-trajectory image generation through image rendering, which can be divided into the following steps.

1). **3D Reconstruction and Pointcloud Alignment**: Use RGB images and corresponding depth map to reconstruct 3D-pointcloud and convert the pointcloud into the coordinate system of ego-vehicle with offset.
2). **Back-projection**: Project the pointcloud onto the image plane of ego-vehicle with offset.
3). **Image Rendering**: Fill the black holes and make the image more consistent.
4). **Crop and Resize**: Crop the edge part of the rendered image into size 400x250 and then resize it to input size of the model (128x128).

The following figure 14-19 shows an example of off-trajectory image generation according to the steps shown above.

Figure 14. Original RGB Image

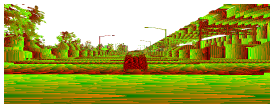

Figure 15. Depth Map

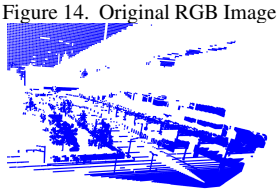

Figure 16. 3D Pointcloud

Figure 17. Back-projection

Figure 18. Rendered Image

Figure 19. Crop and Resized Image

With the image rendering process above, 6,500 samples from 10 trajectories of Town03, 04 are collected from the CARLA simulator as on-trajectory data, and 39,000 samples are generated as off-trajectory data. Then the model with the same architecture as Table IV is trained for around 75 epochs, the validation loss reaches the lowest point 6.4e-3. The losses for Δx, Δy, Δyaw are 1.2e-2, 6.6e-3 and 5e-4 respectively.

The losses were extremely small, especially the loss of Δyaw. The reason might be that the model learns the Δyaw value from the missing parts behind the target vehicle, which are caused when generating off-trajectory images with lateral offsets. The missing parts form a shape similar to the vehicle and always have the same direction as the lateral offset. When having the ego vehicle on the right side of its original position, the missing parts are also on the right side of the target vehicle, vice versa.

This is not the way that we supposed the network to learn the Δyaw value since the missing parts can not be observed in the real images. Although the model may learn the Δyaw value unexpectedly, the model performs well in different test trajectories in town 01, 03, 04, 05. It can also adapt to the different colors of the target vehicle. Also, we tried to implement car platooning with the depth-based model, where there are one target vehicle and more than one ego-vehicles following one after the other. The test videos can be seen in the supplementary link, in the folder */CNN-MLP model with depth camera/*.

Since this CNN model is trained based on images generated with a depth map that is directly collected from depth camera in CARLA simulator, the trained model is called depth-based model in the later comparison.

D. Stereo-based CNN-MLP Model

With the approach described above, offline data collection is already very close to an approach that can be implemented in reality. However, there is still a step in this method that is difficult to achieve in reality. It is hard to obtain such an accurate depth map as Figure 15 from CARLA simulator. [7] has introduced a handheld RGB-D camera, but such RGB-D camera is commonly used for the indoor scene. For the outdoor scene in car-following task, there is no such camera that can generate an accurate depth map.

To solve this problem, one of the most common approaches is stereopsis (stereo vision), in which depths are estimated by triangulation using the two images [8]. The brain uses this binocular disparity to extract depth information from the two-dimensional retinal images which are known as stereopsis. Similarly, here we used a stereo system to estimate the depth map.

The general principle is shown in Figure 20[5]. Two RGB cameras are placed close to each other with a small distance called baseline. For each 3D point X, there are two image coordinates on the corresponding image plane. x and x' represent the distance from the camera center to corresponding image coordinates. And disparity means the difference between x and x' and it refers to the difference in image location of an object seen in two different cameras. The stereo cameras perceive the depth map by using stereo disparity. With disparity map, we can get estimated depth Z with

[5]https://docs.opencv.org/3.4/dd/d53/tutorial_py_depthmap.html

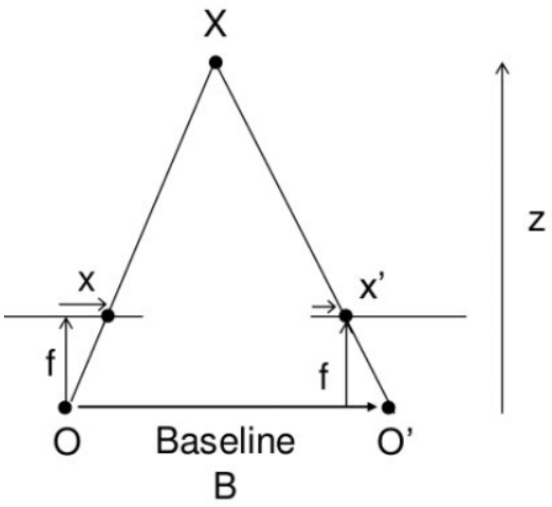

Figure 20. Principle of Stereo Vision System

$$disparity = x - x' = \frac{Bf}{Z} \tag{16}$$

where x and x' are the distance between points in the image plane corresponding to the scene point 3D and their camera center. B is the baseline distance between two cameras (set manually) and f is the focal length of the camera (already known).

With the approach for depth map estimation mentioned above, two RGB-cameras as stereo cameras with a close distance of 0.5m and the y-position of two cameras are y = 0, y = 0.5. Instead of one RGB image and one depth map, two RGB images are collected for each on-trajectory sample. Then OpenCV functions cv.StereoBM_create is used to compute the disparity map and estimate the depth map with the formula above.

The estimated depth map is not so precise, especially in the sky part, which has an extremely large depth. The sky part is white, which means the estimated depth is close to $+\infty$. According to the formula above, this will lead to the black sky part in the back-projection image. To fix this problem, the estimated depth map of the sky part is set to be a fixed limited value, so that the back-projection image can look reasonable and consistent.

Then the depth map estimation can be used to replace the depth map directly obtained from CARLA simulator. Then the off-trajectory images can be generated with the same process as before, including 3D reconstruction, point cloud alignment, back-projection, image rendering.

Figures 21-27 shows an example of off-trajectory image generation based on stereo vision. From the figures can be seen that although many black holes have been filled in after image rendering, there are still many small black holes in the resulting image shown in Figure 27. Since the black holes can not be observed in the real images, they may become noise during training, which will probably affect the training result.

Figure 21. Original Middle RGB Image

Figure 22. Original Right RGB Image

Figure 23. Disparity Map

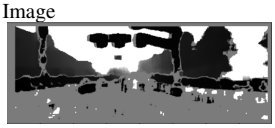

Figure 24. Depth Map Estimation

Figure 25. Back-projected Image with Lateral Offset -2m

Figure 26. Rendered Image with Lateral Offset -2m

Figure 27. Crop and Resized Image with Lateral Offset -2m

Another problem is that the range of possible offset becomes smaller. Only the resulting images with small lateral and rotational offset seem to be generally consistent. However, even extremely small longitudinal offset would lead to large black holes in the resulting images, so no such samples with longitudinal offsets are collected.

Similar to the dataset for the training depth-based model, 8,000 samples from 8 routes in Town 03, 04 are collected as on-trajectory data, and then 32,000 samples with offsets are generated as off-trajectory data. After training for 60 epochs, the validation loss reaches the lowest point 3.9e-2. The losses for Δx, Δy, Δyaw are 2.9e-2, 8.7e-2 and 1.25e-3 respectively, which are much larger than the loss of depth-based model.

It's obvious that the quality of the rendered images, which use the estimated depth map based on a stereo vision system, is not so stable and might contain much noise. So the prediction of control values is not so accurate as of the depth-based model, and the car following performance is not so stable as well in test trajectories in town 01, 03, 04, 05. Also, we tried to implement car platooning with the stereo-based model, where there are one target vehicle and more than one ego-vehicles following one after the other. However, the vehicle following behind might keep shaking because of error accumulation of the preceding vehicle. The test videos can be seen in the supplementary link, in the folder */CNN-MLP model with stereo camera/*.

Since this CNN model is a trained image generated with a

depth map that is estimated through a stereo vision system, the trained model is called a stereo-based model in the later comparison.

VIII. Evaluation and Discussion

A. Evaluation Metrics

Until now, different models with different architectures and datasets have been trained and some of them have a desirable result. But from what perspective can the performance of car following be compared? Especially when the ego-vehicle can follow the target vehicle for the whole trajectory, it's hard for us to qualitatively evaluate the performance. To evaluate the performance of the car following quantitatively, we introduced four evaluation metrics, route completion, infraction count, average translation error, and control difference.

1) ***Route Completion****:* Route Completion(RC) [9] means the ratio between the trajectory that the ego-vehicle can follow and the original trajectory of the target vehicle. Here we compute the ratio in frames. There are two conditions to determine whether the ego-vehicle stopped following the target vehicle:

- The distance between two vehicles is larger than 20m.
- The velocity of ego-vehicle is less than 0.3m/s while the velocity of target vehicle is larger that 1.0m/s.

The reason why we choose 20m as a threshold for stopping following is that when the target vehicle is 20m away in front of the ego-vehicle, the target vehicle is too small to be recognized from the view of ego-vehicle. If there is such a large distance between two vehicles, the ego-vehicle might even lose the target vehicle in its own view.

If any of the conditions are fulfilled and kept for more than 3 frames during car following, it is determined that the ego-vehicle has stopped following. Then the route completion can be computed with

$$RC = \frac{currentframe}{totalframe} \tag{17}$$

We can see from the equation 17 that the larger RC is, the more continuously the ego-vehicle can follow the target and the better performance the model has.

2) ***Infraction Count****:* Infraction Count(IC) [9] represents how stable is the trajectory of ego-vehicle during car following. It can count how many times the ego-vehicle crosses the lane marking and collides with the environment or other objects. Here are the lane invasion detector and the collision detector in CARLA simulator, *sensor.other.lane_invasion and sensor.other.collision* [6] can be used to detect conflicts. While the ego-vehicle keeps following the target vehicle, the infraction count can be computed with

$$IC = 0.8^n * 0.5^m \tag{18}$$

[6]https://carla.readthedocs.io/en/latest/ref_sensors/

where n is the absolute number of lane invasions, m is the absolute number of object collisions. 'Absolute' means that the original conflicts of the target vehicle shouldn't be counted. The number of conflicts should be the number of conflicts between the ego-vehicle and the environment minus the number of conflicts between the target vehicle and the environment. In this way, we can know the larger the result, the less the number of conflicts.

Equation 18 shows that the larger IC is, the more stably the ego-vehicle can follow the target and the better performance the model has. However, when the ego-vehicle stops following the target vehicle according to the two conditions for RC above, the IC will also stop counting. Thus, there is a trade-off between IC and RC. High RC means a longer trajectory to follow, but there might be more conflicts during following, which will lead to a lower IC. Low RC means the ego-vehicle can only follow for a short term, where fewer conflicts would occur, which can bring a higher IC. The model with a high RC and a high IC has the best performance.

3) ***Average Translation Error****:* Average Translation Error (ATE) represents how precisely the ego-vehicle can follow the trajectory of the target vehicle. when the ego-vehicle stops following the target vehicle according to the two conditions for RC above, the trajectory of the ego-vehicle and target vehicle can be matched point-wise. The optimal matching solution with minimum sum of distances can be computed with Hungarian algorithm [10]. Then the relative transformation between the matching points on ego-trajectory and target trajectory can be computed with Equation 19. The relative translations between all pairs of matching points are used to compute ATE with

$$ATE = \frac{1}{|N|}\sum_{i=1}^{N} ||T_i||_2 \tag{19}$$

where $||T_i||_2$ is the second norm of relative translation of each pair of matching points, and N is the number of matching points.

Equation 19 shows that the smaller ATE is, the more precisely the ego-vehicle can recover the trajectory and follow the target and the better performance the model has.

4) ***Control Difference****:* Similar to ATE, control difference (CD) shows how precisely the model can predict the control values at the matching points from the trajectory of the ego-vehicle and the target vehicle. With the matching result through the Hungarian algorithm mentioned above, the L2-loss between the predicted control values and real control values of the target vehicle at each matching point is computed with

$$CD = \frac{1}{|N|}\sum_{i=1}^{N} \sqrt{|a_{ego_i} - a_{target_i}|^2 + |\delta_{ego_i} - \delta_{target_i}|^2} \tag{20}$$

where a_{ego_i} and a_{target_i} are predicted throttle and real throttle of target vehicle. δ_{target_i} and δ_{target_i} are predicted steering angle and real steering angle of target vehicle.

Equation 20 shows that the smaller CD is, the more precise the predicted control values are and the better performance the model has.

B. Comparison between Models

With evaluation metrics described above, the models trained before can be compared. We want to evaluate 4 different models whose descriptions are given below.

- *Basic Model*
  As mentioned in section VII, the basic model has the same structure as Figure 13. The CNN has the architecture shown in Table IV and it is only trained with 16,000 on-trajectory samples from 8 trajectories in town 03, 04. MLP is the model with relative transformation as input, which is described in section V.
- *End-to-end Model*
  As mentioned in section VI, the end-to-end model has architecture shown in Table IV. It is trained end-to-end with around 64,500 samples from 2 trajectories in town 04, including both on-trajectory data and off-trajectory data, which are collected online in CARLA simulator.
- *Depth-based Model*
  As mentioned in section VII-C, depth-based model has the same structure as Figure 13. The CNN has the architecture shown in Table IV. And it is trained with 45,500 samples from 10 trajectories in town 03, 04, including 6,500 on-trajectory samples collected from CARLA simulator and 39 000 off-trajectory samples. The off-trajectory data is generated with a depth map from the depth camera in CARLA simulator. MLP is the model with relative transformation as input, which is described in section V.
- *Stereo-based Model*
  As mentioned in section VII-D, stereo-based model has the same structure as Figure 13. The CNN has the architecture shown in Table IV. And it is trained with 40,000 samples from 8 trajectories in town 03, 04, including 8,000 on-trajectory samples collected from CARLA simulator and 32,000 off-trajectory samples. The off-trajectory data is generated with an estimated depth map based on a stereo vision system. MLP is the model with relative transformation as input, which is described in section V.

With four models to compare, we did inference with them in 10 test trajectories of town 01, 03, 04, 05, which is unseen in the training dataset, and use the average value as an evaluation result. Figure 28 shows the average route completion and infraction count of four different models. And Figure 29 shows the average translation error and control difference of four different models.

From Figure 28 we can find that the end-to-end model can cover most of the target trajectory and have the most stable following behavior. The depth-based model can cover a high ratio of target trajectory but have more conflicts. The basic model and stereo-based model have a low route completion, which means they can only follow the target for a short distance. Despite low route completion, they also have a relatively high infraction count. The performance of the end-to-end model and depth-based model is obviously better than the other two.

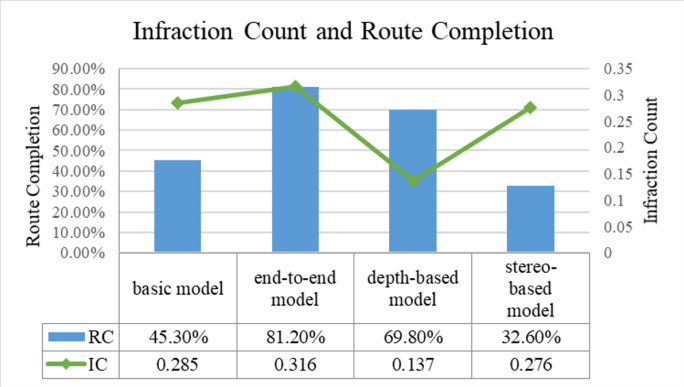

Figure 28. Comparison of IC and RC

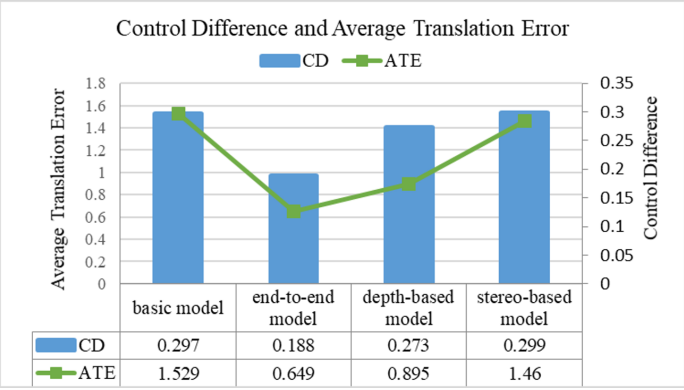

Figure 29. Comparison of ATE and CD

From Figure 29 we can see that the end-to-end model with small average translation error and control difference has the most accurate follow behavior. For the basic model and stereo-based model, the average translation error and control difference is pretty large, which means there is a large deviation between the ego-trajectory and the target trajectory.

In addition, in order to test the robustness against disturbances of different models, we add random disturbances to the ego-vehicle and see how would the route completion change with increasing perturbance strength. In detail, the basic model and depth-based model are tested in 5 trajectories of towns 04 and 05 with four different levels of disturbance. The average values over all 5 trajectories are used as evaluation results. as evaluation result. The random disturbances are implemented with the function *Carla.actor.add_impulse* in CARLA simulator, which can apply an instantaneous force of random direction as an impulse at the center of mass of the actor. Four different levels of impulse strength are 0, 2, 4, 6.

Figure 30 shows how the route completion changes with increasing level of random impulse. First, it is obvious that

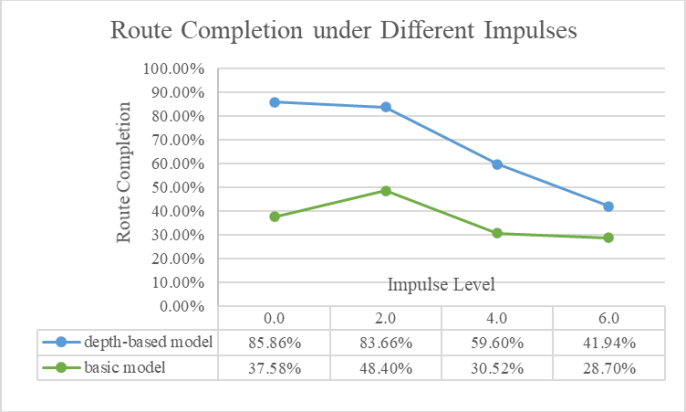

Figure 30. Route Completion of Basic and Depth-based Model under Different Impulses

the depth-based model trained with both on-trajectory and off-trajectory data has higher route completion than the basic model under all levels of impulses, which means the depth-based model has stably better performance. Thus, the data augmentation for off-trajectory data is definitely meaningful.

In addition, we can see from Figure 30, the route completion of both models generally decreases with an increasing level of random impulse. As for the depth-based model, it can still keep a relatively high route completion under the impulse level 2.0. Although the route completion is only around 40% under impulse level 6.0 disturbance, such a large disturbance that could pull the ego-vehicle far away from the original trajectory rarely occurs in reality, so we can think that the depth-based model has certain robustness and good performance.

Through the comparison above, we can draw some conclusions. The model trained end-to-end has the best performance. And the depth-based model can also be used to implement a generally stable car following and its performance is also acceptable. However, due to the undesirable quality of generated images, the stereo-based model has a poor performance, which needs to be further improved.

IX. Conclusion

Through the whole project, we first use model predictive control as a motion planner to give accurate control values, throttle, and steering angle, for car-following task. Then we tried different data collection approaches, including online and offline data collection. With online data collection, MLP models with different inputs, as well as FCNN models are trained end-to-end. To step closer to reality and make the whole process implementable in practice, we started to try offline data collection. First, data augmentation is implemented through strategies like image rendering, stereo-based depth estimation. Then CNN-MLP models are trained with different datasets.

The end-to-end model with a dataset collected directly from CARLA simulator has the best performance. The depth-based model also has a desirable performance. They can also be used to implement car platooning. However, the performance of the stereo-based model, which is the method that is most likely to be implemented in reality, is not satisfactory by comparison. The most possible reason is that the quality of the augmented data based on stereo vision system needs to be further improved.

From this point of view, there are still many possibilities to work further in the future. Since the estimated depth map based on a stereo vision system is not accurate, we can also try to train a model with auto-encoder-like architecture for depth map prediction instead of the stereo vision system. In addition, so far all models require the velocities of both vehicles as input, but it is hard to measure the exact velocity of the target vehicle in practice. So another idea is to use an RNN (recurrent neural network) to predict the velocity of the target vehicle based on a sequence of images.

Acknowledgment

We would like to thank our tutors, Qadeer Khan and Mariia Gladkova who guided us in doing the projects. They provided us with valuable advice and helped us in difficult periods. Their motivation and help contributed tremendously to the successful completion of the project.

References

[1] P. Kavathekar and Y. Chen, "Vehicle platooning: A brief survey and categorization," in *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, vol. 54808, 2011, pp. 829–845.

[2] T. Robinson, E. Chan, and E. Coelingh, "Operating platoons on public motorways: An introduction to the sartre platooning programme," in *17th world congress on intelligent transport systems*, vol. 1, 2010, p. 12.

[3] R. Kianfar, P. Falcone, and J. Fredriksson, "A control matching model predictive control approach to string stable vehicle platooning," *Control Engineering Practice*, vol. 45, pp. 163–173, 2015.

[4] V. Turri, Y. Kim, J. Guanetti, K. H. Johansson, and F. Borrelli, "A model predictive controller for non-cooperative eco-platooning," in *2017 American Control Conference (ACC)*, 2017, pp. 2309–2314.

[5] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "Carla: An open urban driving simulator," in *Conference on robot learning*. PMLR, 2017, pp. 1–16.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[7] F. Steinbrücker, J. Sturm, and D. Cremers, "Real-time visual odometry from dense rgb-d images," in *2011 IEEE international conference on computer vision workshops (ICCV Workshops)*. IEEE, 2011, pp. 719–722.

[8] A. Saxena, J. Schulte, A. Y. Ng *et al.*, "Depth estimation using monocular and stereo cues." in *IJCAI*, vol. 7, 2007, pp. 2197–2203.

[9] A. Prakash, K. Chitta, and A. Geiger, "Multi-modal fusion transformer for end-to-end autonomous driving," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 7077–7087.

[10] G. A. Mills-Tettey, A. Stentz, and M. B. Dias, "The dynamic hungarian algorithm for the assignment problem with changing costs," *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-07-27*, 2007.